Amortised Memory Analysis Using the Depth of Data Structures

Amortised Memory Analysis using the Depth of Data Structures

Brian Campbell

School of Informatics, University of Edinburgh
Brian.Campbell@ed.ac.uk*

Abstract. Hofmann and Jost have presented a heap space analysis [1] that finds linear space bounds for many functional programs. It uses an amortised analysis: assigning hypothetical amounts of free space (called potential) to data structures in proportion to their sizes using type annotations. Constraints on these annotations in the type system ensure that the total potential assigned to the input is an upper bound on the total memory required to satisfy all allocations.
We describe a related system for bounding the stack space requirements which uses the depth of data structures, by expressing potential in terms of maxima as well as sums. This is achieved by adding extra structure to typing contexts (inspired by O'Hearn's bunched typing [2]) to describe the form of the bounds. We will also present the extra steps that must be taken to construct a typing during the analysis.

Obtaining bounds on the resource requirements of programs can be crucial for ensuring that they enjoy reliability and security properties, particularly for use in constrained systems such as mobile phones, smartcards and embedded systems. Hofmann and Jost have presented a type-based amortised analysis for finding upper bounds on the heap memory required for programs in a simple functional programming language [1]. The form of these bounds is limited to linear functions with respect to the size of the program's input. Fortunately, this is sufficient for a wide variety of interesting programs. Moreover, the analysis was successfully used to certify such bounds in a Proof Carrying Code system [3].

However, it is also important to bound the stack space requirements, especially for functional programs where it is easy to cause excessive stack usage by accident. The Hofmann-Jost analysis has previously been adapted to measure stack space [4, 5], but the form of the bounds was again limited to linear functions in terms of the *total* size of the input.

In this work we present a similar analysis where bounds are given as max-plus expressions on the *depth* of data structures. This is far more precise for programs operating on tree-structured data.

Like Hofmann-Jost, our analysis consists of two parts: a type-system which *certifies* that a given bound really is an upper bound on the stack memory

* This work was partially supported by the ReQueST grant (EP/C537068/1) from the Engineering and Physical Sciences Research Council.

requirements, and an inference procedure based on Linear Programming for that type system. In our type system we impose extra structure on the typing contexts to represent the form of the bounds (where to take the maximum and where to add), and hence we use extra structural typing rules to manipulate the context. We also add an extra stage to the inference to determine where these structural rules should be used.

We begin by presenting the programming language and its metered operational semantics, then in Section 2 give details of the type system and consider some examples of typings in Section 3. We prove that the type system correctly certifies bounds in Section 4 before presenting the inference procedure in Section 5. In Section 6 we discuss some limitations when analysing programs with nested datatypes. Finally we describe some extensions to the analysis (Section 7) and related work (Section 8).

1 Language and operational semantics

We consider a simple first-order call-by-value functional programming language. The syntax of the language is presented in Figure 1, where programs P are given as a sequence of function definitions D with expressions e. For brevity's sake we only consider computations on units ($*$), booleans, pairs, sums and binary trees. The syntax requires programs to be in a 'let-normal' form which makes the evaluation order explicit by requiring variables rather than subexpressions in various places. We will discuss extensions to the language in Section 7.

$$
\begin{aligned}
P &:= \mathsf{let}\ D \mid \mathsf{let}\ D\ P \\
D &:= f(x_1, \ldots, x_p) = e_f \\
e &:= * \mid \mathsf{true} \mid \mathsf{false} \mid x \mid f(x_1, \ldots, x_p) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \\
&\quad \mid (x_1, x_2) \mid \mathsf{match}\ x\ \mathsf{with}\ (x_1, x_2) \to e \\
&\quad \mid \mathsf{inl}(x) \mid \mathsf{inr}(x) \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{inl}(x_l) \to e_l \mid \mathsf{inr}(x_r) \to e_r \\
&\quad \mid \mathsf{leaf} \mid \mathsf{node}(x_l, x_r, x_v) \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{leaf} \to e_1 \mid \mathsf{node}(x_l, x_r, x_v) \to e_2
\end{aligned}
$$

Fig. 1. Syntax

We have a large step operational semantics for the language, which includes metering of the free stack space. Values in the language are units, booleans, pairs, sums and heap locations for trees, with a distinguished location null which represents leaf. A selection of the rules for the operational semantics appear in Figure 2. The judgements have the form

$$m, S, \sigma \vdash e \rightsquigarrow v, \sigma'$$

meaning that with m units of stack space, an environment S mapping variable names to values and a store σ mapping non-null locations to binary tree triplets,

$$\frac{S(x_1) = v_1 \ \ldots \ S(x_p) = v_p \qquad m, [y_1 \mapsto v_1, \ldots, y_p \mapsto v_p], \sigma \vdash e_f \rightsquigarrow v, \sigma' \quad \text{the } y_i \text{ are the symbolic arguments in the definition of } f}{m + \mathrm{stack}(f), S, \sigma \vdash f(x_1, \ldots, x_p) \rightsquigarrow v, \sigma'} \text{ E-Fun}$$

$$\frac{m, S, \sigma \vdash e_1 \rightsquigarrow v_0, \sigma_0 \qquad m, S[x \mapsto v_0], \sigma_0 \vdash e_2 \rightsquigarrow v, \sigma'}{m, S, \sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v, \sigma'} \text{ E-Let}$$

$$\frac{}{m, S, \sigma \vdash \mathsf{leaf} \rightsquigarrow \mathsf{null}, \sigma} \text{ E-Leaf}$$

$$\frac{\sigma' = \sigma[l \mapsto \langle S(x_l), S(x_r), S(x_v)\rangle] \qquad l \notin \mathrm{dom}(\sigma)}{m, S, \sigma \vdash \mathsf{node}(x_l, x_r, x_v) \rightsquigarrow l, \sigma'} \text{ E-Node}$$

$$\frac{S(x) = \langle v_l, v_r, v_v\rangle \qquad m, S[x_l \mapsto v_l, x_r \mapsto v_r, x_v \mapsto v_v], \sigma \vdash e_2 \rightsquigarrow v, \sigma'}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{leaf} \to e_1 \mid \mathsf{node}(x_l, x_r, x_v) \to e_2 \rightsquigarrow v, \sigma'} \text{ E-MatchNode}$$

Fig. 2. Sample rules from the operational semantics

the expression e can evaluate to the value v with the new store σ'. We do not need to include the amount of stack space afterwards because the stack discipline will ensure that it is m again (this is easily checked in the full set of rules).

Note that we assume that stack space is allocated one frame at a time on function entry, and released on exit. We denote the size of frame required by function f by $\mathrm{stack}(f)$. We expect that our techniques could also be applied to more fine grained stack machines. Indeed, a simpler Hofmann-Jost style stack space analysis has been applied to a stack machine cost model for the Hume language [4].

We will also need to mention an unmetered form of the operational semantics. For this we simply drop the m part of each judgement.

Example 1. The andtrees function computes the point-wise boolean 'and' of two binary trees with boolean values at the nodes:

```
let andtrees(t1,t2) =
  match t1 with leaf → leaf | node(l1,r1,v1) →
  match t2 with leaf → leaf | node(l2,r2,v2) →
    let l = andtrees(l1,l2) in
    let r = andtrees(r1,r2) in
    let v = if v1 then v2 else false in
      node(l,r,v)
```

This function requires at most $\mathrm{stack}(\mathrm{andtrees}) \times (\min\{|\mathrm{t1}|_d, |\mathrm{t2}|_d\} + 1)$ units of stack space to run (where depth $|\cdot|_d$ is defined by $|\mathsf{null}|_d = 0$ and $|l|_d = 1 + \max\{|l_1|_d, |l_2|_d\}$ when $\sigma(l) = \langle l_1, l_2, v\rangle$).

2 Type system

We now describe the type system that can be used to provide certified bounds. The key notion in the type system is that the function from input size to the stack space bound is encoded by annotations in the types and the structure of the typing context. This is similar to the 'physicist's view' of amortised analysis described by Tarjan [6]. Following Tarjan, we call this assignment *potential*.

The types and contexts are given by

$$T := 1 \mid \mathsf{bool} \mid T_1 \otimes T_2 \mid (T_1, k_1) + (T_2, k_2) \mid \mathsf{tree}(T, k).$$
$$\Gamma := \cdot \mid x : T \mid k \mid \Gamma_1, \Gamma_2 \mid \Gamma_1; \Gamma_2.$$

where the annotations k are positive rational numbers. For sum types the annotations represent different contributions to the bound depending upon the choice, and for trees the annotations represent a requirement of k times the depth of the tree (not counting the leaves). Fractional amounts are allowed; for instance, a tree with annotation of one half corresponds to one unit of stack space for every second level of the tree.

The typing contexts have two context formers: one for summing the contribution of the subcontexts (,) and one for taking the maximum (;). For example, the context

$$(x : \mathsf{tree}(\mathsf{bool}, 5); y : \mathsf{tree}(\mathsf{bool}, 3)), 6$$

represents the bound

$$\max\{5 \times |x|_d, 3 \times |y|_d\} + 6.$$

Thus our typing contexts take the form of trees. To allow a greater range of bounding functions to be represented we also allow variables to appear several times in the context, so long as the underlying types (but not necessarily the annotations) are the same. We will implicitly take ',' and ';' to be associative throughout.

The formal encoding of this potential in types and typing contexts is given as the Υ_{t} and Υ_{c} functions in Figure 3.

Using structured contexts in this way was inspired by O'Hearn's Bunched Typing [2], where a typical application of the structure was to denote heap separation of data structures with one context former, and possible sharing of heap data with the other.

We represent function signatures as a map Σ from function names to a signature $\Gamma \to T, k$ where Γ is a context containing each parameter once, T is the result type and k the fixed amount of potential to add to that from T.

The type system has two groups of rules. The syntax-directed rules feature side conditions which ensure that the potential of the context is a sufficient amount of stack space to evaluate the expression, *and* that there is enough potential in the context to account for all of the potential in the result type (we will make this more formal in Theorem 1, below). The latter requirement is needed to translate bounds for subsequent parts of the program — the typing of these later parts may give a bound in terms of the size of the result of this

$$
\begin{aligned}
\Upsilon_{\mathsf{t}}(\sigma, *, 1) = \Upsilon_{\mathsf{t}}(\sigma, \mathsf{true}, \mathsf{bool}) &= \Upsilon_{\mathsf{t}}(\sigma, \mathsf{false}, \mathsf{bool}) = 0 \\
\Upsilon_{\mathsf{t}}(\sigma, (v', v''), T' \otimes T'') &= \Upsilon_{\mathsf{t}}(\sigma, v', T') + \Upsilon_{\mathsf{t}}(\sigma, v'', T''), \\
\Upsilon_{\mathsf{t}}(\sigma, \mathsf{inl}(v), (T', k') + (T'', k'')) &= k' + \Upsilon_{\mathsf{t}}(\sigma, v, T'), \\
\Upsilon_{\mathsf{t}}(\sigma, \mathsf{inr}(v), (T', k') + (T'', k'')) &= k'' + \Upsilon_{\mathsf{t}}(\sigma, v, T''), \\
\Upsilon_{\mathsf{t}}(\sigma, \mathsf{null}, \mathsf{tree}(T, k)) &= 0 \\
\Upsilon_{\mathsf{t}}(\sigma, l, \mathsf{tree}(T, k)) &= \max\{\Upsilon_{\mathsf{t}}(\sigma, v_l, \mathsf{tree}(T, k)), \Upsilon_{\mathsf{t}}(\sigma, v_r, \mathsf{tree}(T, k)), \Upsilon_{\mathsf{t}}(\sigma, v_v, T)\} + k \\
&\qquad \text{where } \sigma(l) = \langle v_l, v_r, v_v \rangle.
\end{aligned}
$$

$$
\begin{aligned}
\Upsilon_{\mathsf{c}}(\sigma, S, \cdot) &= 0, \\
\Upsilon_{\mathsf{c}}(\sigma, S, x : T) &= \Upsilon_{\mathsf{t}}(\sigma, S(x), T), \\
\Upsilon_{\mathsf{c}}(\sigma, S, k) &= k, \\
\Upsilon_{\mathsf{c}}(\sigma, S, (\Gamma, \Delta)) &= \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) + \Upsilon_{\mathsf{c}}(\sigma, S, \Delta), \\
\Upsilon_{\mathsf{c}}(\sigma, S, (\Gamma; \Delta)) &= \max\{\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma), \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)\}.
\end{aligned}
$$

Fig. 3. Assignment of potential to values and environments according to their types and contexts

expression, and we wish to translate it into a bound with respect to the size of the input values *only*. Thanks to this translation we do not need a separate size analysis to give a relationship between the size of the intermediate values and the input.

However, the syntax-directed rules require the typing context to have a specific structure. To manipulate the context structure to fulfil these requirements we also have a set of *structural* rules.

A representative sample of the syntax-directed rules is given in Figure 4 and all of the structural rules are given in Figure 5. The typing judgements take the form

$$\Gamma \vdash e : T, k'$$

meaning that in the context Γ the expression e can be given type T, and the potential assigned to the result is given by the annotations in T *plus* the fixed amount k'. In some places we use $\Gamma()$ to denote a context with a hole, and $\Gamma(\Delta)$ when that hole is filled by Δ. We write $q \times \Gamma$ to denote the context Γ with every annotation k replaced by qk.

The VAR and LEAF rules are the simplest: evaluation of the expressions requires no stack space and we only need to ensure that the potential of the result, k', is accounted for by a fixed amount in the context, k. Note that the annotation for the leaf's type, k_1, can be anything because we consider the depth of a leaf to be zero. The NODE rule's side condition requires an extra k_1 units because the resulting tree is one level deeper than the larger subtree.

The FUN rule has two side conditions. The first ensures that there is enough potential in the context to account for the allocation of a stack frame for the

$$\frac{k \geq k'}{x : T, k \vdash x : T, k'}\ \textsc{Var} \qquad \frac{k \geq k'}{\cdot, k \vdash \mathsf{leaf} : \mathsf{tree}(T, k_1), k'}\ \textsc{Leaf}$$

$$\frac{k \geq \mathrm{stack}(f) \qquad k + k_1' \geq k' \qquad \Sigma(f) = \Gamma \to T, k_1' \qquad (y_1, \ldots, y_p) = \mathrm{names}(\Gamma)}{\Gamma[x_1/y_1, \ldots, x_p/y_p], k \vdash f(x_1, \ldots, x_p) : T, k'}\ \textsc{Fun}$$

$$\frac{\Delta \vdash e_1 : T_0, k_0 \qquad \Gamma(x : T_0, k_0) \vdash e_2 : T, k'}{\Gamma(\Delta) \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T, k'}\ \textsc{Let}$$

$$\frac{k \geq k_1 + k'}{(x_l : \mathsf{tree}(T, k_1); x_r : \mathsf{tree}(T, k_1); x_v : T), k \vdash \mathsf{node}(x_l, x_r, x_v) : \mathsf{tree}(T, k_1), k'}\ \textsc{Node}$$

$$\frac{\Gamma(\cdot) \vdash e_1 : T, k' \qquad \Gamma((x_l : \mathsf{tree}(T, k_1); x_r : \mathsf{tree}(T, k_1); x_v : T), k_1) \vdash e_2 : T, k'}{\Gamma(x : \mathsf{tree}(T, k_1)) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{leaf} \to e_1 \mid \mathsf{node}(x_l, x_r, x_v) \to e_2 : T, k'}\ \textsc{Match}$$

Fig. 4. Sample syntax-directed typing rules

callee, and the second gives the two possible sources for 'translating' k', either from the potential k used to show that we can allocate the stack frame, or from the amount k_1' given by the result of the callee (which must ultimately have come from somewhere in Γ).

The Let rule is more subtle. Intuitively, we can just take the maximum of the stack space bounds for e_1 and e_2, but we must also consider how to translate the parts of e_2's bound that are expressed in terms of the size of the value of the bound variable x. Hence we locally replace the part of the context used for e_1 with x, which allows for both the stack allocation required for e_1 and the translation of subsequent requirements in e_2 given in terms of the size of x. To be sound this requires that all of the allocations that we consider respect the stack discipline; that is, all allocations in e_1 are deallocated before the evaluation of e_1 is complete. If we consider memory allocations in e_1 that may persist into e_2 (such as heap memory) then we may not have the free memory 'promised' by the annotations in the surrounding context, $\Gamma()$.

The Match rule uses a similar local replacement, but this is simply an unfolding of the tree structure into the context and does not require the stack discipline for soundness.

The structural rules allow the manipulation of contexts to fit the requirements of the syntax-directed rules. The two weakening rules remove sections of the context and unnecessary potential. The context equivalence rule $\equiv$ replaces part of the typing context with one whose contents and potential are identical, using any of the equivalences from Figure 6. Note that all of these equivalences are reversible.

The plus-contract case of $\equiv$ illustrates an important difference from the Hofmann-Jost heap analysis. In that system contraction treated the annotations of nested types (such as $\mathsf{tree}(\mathsf{tree}(\mathsf{bool}, 3), 2)$) independently. Here we can only uniformly scale the entire context. This restriction is necessary because we

$$\frac{\Gamma(\Delta) \vdash e : T, k'}{\Gamma(\Gamma'(\Delta)) \vdash e : T, k'} \textsc{Weaken}$$

$$\frac{\Gamma(x : T[k_1/k]) \vdash e : T', k' \qquad k \geq k_1}{\Gamma(x : T) \vdash e : T', k'} \textsc{WeakenA}$$

$$\frac{\Gamma(\Delta') \vdash e : T, k' \qquad \Delta \cong \Delta'}{\Gamma(\Delta) \vdash e : T, k'} \equiv$$

$$\frac{\Gamma(q \times \Delta, (1-q) \times \Delta') \vdash e : T, k' \qquad q \in [0,1]}{\Gamma(\Delta; \Delta') \vdash e : T, k'} \textsc{Split}$$

Fig. 5. Structural typing rules

are measuring the depth of the entire data structure weighted by the annotations and treating the annotations independently can change the ratio of the weightings and hence may alter which path through the data structure is the 'deepest'. Uniform scaling maintains the deepest path, ensuring that the potential does not change as a result of applying the typing rule.

$$\begin{array}{rlrl}
\Gamma, \Delta \cong \Delta, \Gamma & \text{(plus-commute)} & \Gamma; \Delta \cong \Delta; \Gamma & \text{(max-commute)} \\
\Gamma, (\Delta; \Delta') \cong (\Gamma, \Delta); (\Gamma, \Delta') & \text{(distribute)} & \Gamma \cong \Gamma; \Gamma & \text{(max-contract)} \\
\Gamma \cong \Gamma, \cdot & \text{(plus-empty)} & \Gamma \cong \Gamma; \cdot & \text{(max-empty)} \\
\Gamma \cong \Gamma, 0 & \text{(plus-zero)} & \Gamma \cong \Gamma; 0 & \text{(max-zero)} \\
\Gamma \cong \Delta \quad \text{if} \quad \Delta \cong \Gamma & \text{(symmetry)} & & \\
\multicolumn{4}{c}{\Gamma \cong q \times \Gamma, (1-q) \times \Gamma \quad \text{for } q \in [0,1] \qquad \text{(plus-contract)}}
\end{array}$$

Fig. 6. Equivalent contexts (for the $\equiv$ typing rule)

Split is the typing rule of last resort — it approximates a bound given as a sum by a bound given as a maximum. For example, when $q = 1/2$ and the potential of Δ and Δ' is given by x and y, Split corresponds to the fact that

$$\forall x, y \in \mathbb{Q}^+, \quad \max\{x, y\} \geq x/2 + y/2,$$

where $\mathbb{Q}^+$ is the set of rationals greater than or equal to zero. The set of inequalities corresponding to Split are the best we can give without requiring extra information about x and y. Split is useful in two places; during inference when conflicting structural requirements force the approximation, and when one of the subcontexts can be ignored for the purposes of giving a bound (for example, because it is a boolean). In the latter case q is 0 or 1 and there is no approximation.

We also give the following two derived rules to manipulate fixed amounts of potential:

$$\frac{\Gamma(k_1, k_2) \vdash e : T, k' \qquad k = k_1 + k_2}{\Gamma(k) \vdash e : T, k'}\ \textsc{ContractA}$$

$$\frac{\Gamma(k) \vdash e : T, k' \qquad k = k_1 + k_2}{\Gamma(k_1, k_2) \vdash e : T, k'}\ \textsc{ContractA}'$$

These two contraction rules replace most of the cases of plus-contract where the factor q is not known. Thus if we fix a value for q in the remaining cases and the uses of Split we will only have linear equalities and inequalities as side conditions, which will allow us to use Linear Programming during the inference.

3 Examples of typing derivations

Before proving the soundness of the type system we consider some examples.

Example 1 (Continued). The precise bound given in Section 1 contained a minimum, whereas we only consider max-plus bounds. Hence we hope to show that

$$\text{stack(andtrees)} \times (1 + |\text{t1}|_d) \quad \text{and} \quad \text{stack(andtrees)} \times (1 + |\text{t2}|_d)$$

are bounds on the stack space required. These correspond to the following two type signatures:

$$\text{t1} : \mathsf{tree}(\mathsf{bool}, \text{stack(andtrees)}), \text{t2} : \mathsf{tree}(\mathsf{bool}, 0) \to \mathsf{tree}(\mathsf{bool}, \text{stack(andtrees)}), 0$$
$$\text{t1} : \mathsf{tree}(\mathsf{bool}, 0), \text{t2} : \mathsf{tree}(\mathsf{bool}, \text{stack(andtrees)}) \to \mathsf{tree}(\mathsf{bool}, \text{stack(andtrees)}), 0$$

Note that these signatures do not include the last stack(andtrees) units of space; this is added by the typing of the function call in the caller. The annotation on the result says that the bound is also at least stack(andtrees) times the depth of the result, too.

We can obtain either of these signatures with a type derivation of the structure shown in Figure 7, where $k_1 = \text{stack(andtrees)}, k_2 = 0$ for the first bound, and vice versa for the second. The only non-trivial side condition in this typing is $k_1 + k_2 = k$ from ContractA′. Note that the uses of Let in the derivation focus in on exactly the subcontext required for the recursive calls and the calculation of v.

Example 2. Using maxima in the potential also allows more precise bounds to be derived than plain Hofmann-Jost is able to. Consider the following function:

```
let maybeleft(t,b) =
  match t with leaf → leaf | node(l,r,v) →
    if b then l else t
```

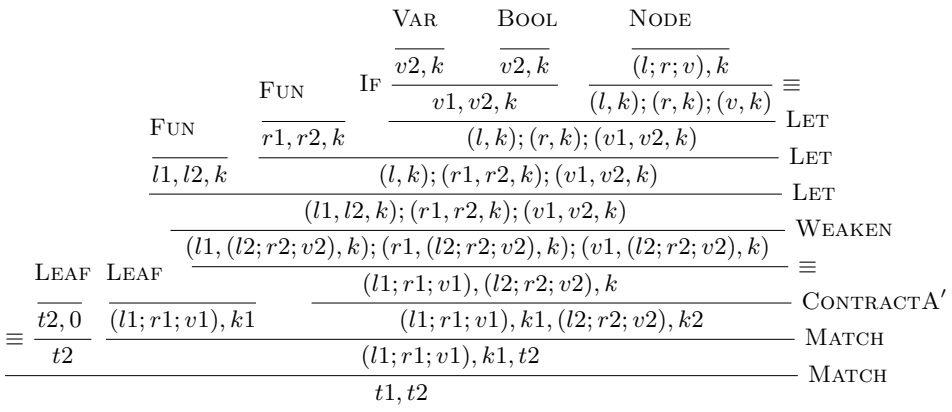

Fig. 7. Typing derivation structure and contexts for andtrees example

While the function itself only requires a constant amount of stack space, its typing is used to translate bounds in terms of its result's size into bounds in terms of t. In the present system we can obtain the signature

$$\mathsf{tree}(T, k); \mathsf{bool} \to \mathsf{tree}(T, k), 0$$

indicating that the size of t is an upper bound on the size of the result. The key part of the typing is that we can use the max-contract form of the $\equiv$ rule to take the maximum of the two branches of the if expression. However, other Hofmann-Jost analyses can only sum the potential for the branches of the if, which doubles the part of the bound expressed in terms of t's size.

4 Soundness

Our main result is that any amount of potential that we can assign to a typing context and still type an expression using it is a sufficient amount of stack space to evaluate that expression. For the induction, we also show that the potential assigned to the result is at most the amount we began with and that any extra space q is preserved.

Theorem 1. *If an expression e in some well-typed program has a typing*

$$\Gamma \vdash e : T, k'$$

and an evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ and $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ is defined, then for any $q \in \mathbb{Q}^+$ and $m \in \mathbb{N}$ such that

$$m \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Gamma) + q$$

m will be a sufficient amount of stack space for the execution to succeed,

$$m, S, \sigma \vdash e \rightsquigarrow v, \sigma',$$

and

$$m \geq \Upsilon_{\mathsf{t}}(\sigma', v, T) + k' + q.$$

Proof. (Sketch.) We proceed by simultaneous induction on the evaluation and the typing derivations. First, note that whenever we use a value from S or σ we can be sure that it has the expected form for its type because otherwise $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ would not be defined.

For the leaf evaluation rules (E-Leaf, E-Node and their unit, boolean, sum and pair counterparts) no extra stack memory is required so the execution will always succeed. It is then sufficient to check that $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ plus the given side condition is at least the potential assigned to the result.

The other rules need to use the induction hypothesis. The precondition on m can be satisfied by showing that the original $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma)$ is larger than or equal to its counterpart for the induction hypothesis. The result of the induction hypothesis is sufficient for most of the rules, where the resulting value and type from the induction hypothesis are also the value and type of the current expression. The Fun rule is a little different due to the stack space used, and Let rule uses the induction hypothesis twice. As these are the most interesting cases, we consider them in a little more detail.

Fun. As the entire program is well-typed there must be a typing of the function body:

$$\Gamma \vdash e_f : T, k_1'.$$

Now we can check that m is sufficient for both the allocation and the induction hypothesis (note that the side condition guarantees that $q + k - \mathrm{stack}(f)$ is positive):

$$\begin{aligned} m &\geq \Upsilon_{\mathsf{c}}(\sigma, S, (\Gamma[x_1/y_1, \ldots, x_p/y_p], k)) + q \\ &= \Upsilon_{\mathsf{c}}(\sigma, [y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \Gamma) + k + q \\ &\geq \mathrm{stack}(f) + \Upsilon_{\mathsf{c}}(\sigma, [y_1 \mapsto S(x_1), \ldots, y_p \mapsto S(x_p)], \Gamma) + (q + k - \mathrm{stack}(f)). \end{aligned}$$

From the induction hypothesis we also have

$$\begin{aligned} m &\geq \mathrm{stack}(f) + \Upsilon_{\mathsf{t}}(\sigma', v, T) + k_1' + (q + k - \mathrm{stack}(f)) \\ &\geq \Upsilon_{\mathsf{t}}(\sigma', v, T) + k' + q, \end{aligned}$$

as required.

Let. It can be easily shown that $\Upsilon_{\mathsf{c}}(\sigma, S, \Gamma(\Delta)) \geq \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$, which allows us to apply the induction hypothesis to e_1.

We can also use the induction hypothesis to deduce that the potential has not increased. If we set $m_1 = \lceil \Upsilon_{\mathsf{c}}(\sigma, S, \Delta) \rceil$ and $q = m_1 - \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$ we can see that $m_1 - q = \Upsilon_{\mathsf{c}}(\sigma, S, \Delta)$, and from the induction hypothesis we know that $m_1 - q \geq \Upsilon_{\mathsf{t}}(\sigma_0, v_0, T_0) + k_0$. Thus,

$$\Upsilon_{\mathsf{c}}(\sigma, S, \Delta) \geq \Upsilon_{\mathsf{t}}(\sigma_0, v_0, T_0) + k_0.$$

Thus we can also establish that m is sufficient to apply the induction hypothesis to e_2. □

5 Checking and Inference

For type checking we assume that we are given the full typing derivation (in the implementation we use an assignment of extra terms to uses of the structural typing rules), including rational values for the annotations. It then remains to check that each arithmetic side condition is satisfied.

Our inference procedure has three main steps:

1. Construct a plain (that is, unannotated) typing;
2. add the context structure and uses of the structural rules to obtain a typing derivation in the system of Section 2, modulo side conditions; then
3. use standard Linear Programming techniques such as the Simplex method to solve the side conditions and minimise the bound.

The first stage can be performed by standard unification methods. The middle stage is new to this analysis; previous Hofmann-Jost systems had relatively little extra structure that might not be present in a plain typing (the exception to this is contraction, which must be explicit in order to sum all the requirements).

To make the inference more tractable we assume that the user provides the *structure* of the function signatures, but they need not give actual values for the annotations. For instance, in the andtrees examples we could supply the structure

$$\mathrm{t1}, \mathrm{t2}$$

without specifying the types or values for their annotations.

With the typing from the first stage and the structure for function signatures we can derive a 'desired' typing context for each leaf in the typing derivation. For example, the construction of the new node in the andtrees example has a desired context of

$$(\mathrm{l} : \mathsf{tree}(\mathsf{bool}, k_1); \mathrm{r} : \mathsf{tree}(\mathsf{bool}, k_1); \mathrm{v} : \mathsf{bool})\,, k$$

to match the NODE rule. Note that k_1 and k are just symbolic annotations; the actual values are determined by the Linear Programming solver in the final stage.

We then work outwards in the typing derivation until we have a 'desired' context for the entire function body. For example, if we have an expression $\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ and desired contexts Γ_1 and Γ_2 for e_1 and e_2 respectively, then we can take $\Gamma_1; \Gamma_2; x$ (the maximum bound of the branches) as the desired context for the entire expression. Note that we need to add a use of WEAKEN to the typing derivations for e_1 and e_2 to remove the irrelevant subcontexts.

Binding constructs are more challenging. To simplify the problem we note that we can use the $\equiv$ typing rule to expand a context into a maximum of sums form, and also contract an expanded context into its original form. Once we have the contexts for the subexpressions in this form we can factor out the bound variables and use CONTRACTA$'$ to split any fixed amount k between the part of the context changed by the expression's typing rule and the surrounding context (denoted $\Gamma()$ in the typing rules). It is during this factoring that we

may need to introduce an approximation using Split. Finally, the expression's typing rule provides us with the desired typing context for the whole expression.

We must also add structural rules to bridge any gap between the desired context inferred for the function body and the given function signature. Fortunately this can be treated as an extreme form of binding construct, where every variable is bound. Should the desired context have any symbolic annotation k without a corresponding source in the function signature the plus-zero or max-zero cases of $\equiv$ can be used to fix k to be zero.

Finally, we gather the side conditions from the resulting typing and use a Linear Programming solver to minimise the overall bound. This step may fail if the resource usage is super-linear, or too subtle for the analysis (for example, because it relies on some unmodelled invariant).

Applying the inference procedure to the examples in Section 3 yields the same bounds as our manual use of the type system. However, the derivations are more verbose, mostly due to the context expansion at every binding construct.

The extra stage in the inference also adds to the amount of work the inference performs. In the worst case the expansion can be exponential with respect to the context size, but in practice the execution times remain similar to earlier Hofmann-Jost analyses [7, Appendix B].

6 Containers

Nested data structures such as trees of trees can present a problem for the analysis. The limitation is that we always take the depth of the *entire* data structure, including all nested contents. In a tree of trees this is the longest path (weighted by the annotations) from the root of the outer tree to a leaf of the inner trees. Hence when we move values around the outer tree (to sort it, for example) we may change the overall depth despite leaving the depth of the outer tree and each inner tree alone.

Ideally we would wish to express the overall 'size' of the data structures differently; namely, as the depth of the outermost structure, plus the maximum depth of the structures in the next layer, and so on for further nested layers: essentially, the sum of the maximum depth of each layer. We conjecture that the present type system could be extended in this direction by allowing constrained movement of 'contents' variables in the context to mimic the corresponding movement of values in their container. However, we leave this to future work.

For containers with simpler contents which carry no potential (units, booleans and pairs thereof in the above language) we can adopt a simpler solution. As these values are assigned no potential, there is no approximation involved in using the Split rule with $q = 0$ to 'lift' these variables to the outermost level of the context. Then the inference procedure described above is able to use them wherever necessary.

We have successfully applied this technique to infer bounds on a functional heap sort, and in particular it can show that the internal routines in the sort use stack space proportional to the heap depth.

7 Extensions

The analysis can be extended in several ways. Algebraic datatypes can be incorporated by assigning structured contexts to constructors in a similar manner to function signatures and generalising the typing rules for trees. These structures can be derived automatically to provide bounds with respect to depth, or left to the user for greater flexibility. For example, two forms of product can be defined using these datatypes: a plus-product that behaves as the product presented above, and a max-product where the maximum of the potential of the two values is taken.

The *resource polymorphism* extension hypothesised in the conclusions to Hofmann and Jost's paper [1] can also be applied to allow different function signatures to be inferred for different uses of a function, reflecting the local resource requirements. Tail call optimisation can also be taken into account, and the soundness proof can be extended to partial evaluations of non-terminating programs.

Details of these extensions, including a full soundness proof and formal details of the inference procedure can be found in the author's thesis [7, Chapters 6 and 7]. The extended type checker and inference procedures have been implemented in Standard ML and are available online[1] along with a small selection of example programs.

8 Related work

We have already mentioned some of the recent work on Hofmann-Jost, notably the extension to the Hume language in the Embounded project [4, 8]. Jost has also worked on extensions for higher-order functions [9] and object oriented programming and mutable references [10], although there is not currently an inference procedure for the latter. These features are largely orthogonal to our work, as they change the language but only infer linear total size bounds. Indeed, one possible avenue of future work is to apply our techniques to these analyses.

Most other analyses are based upon some form of sized-types. An early example is Reistad and Gifford's system for finding execution time estimates to assist parallelisation [11], although they avoid detailed analysis of recursive functions by providing fixed types for a small range of library functions such as `map` and `fold` instead. Hughes and Pareto used their sized-types work to certify heap and stack space bounds in a first-order 'embedded ML' language [12], but do not provide any inference.

A strand of inference work on sized-types systems starts with Chin and Khoo's inference [13]. Like Pareto's checker, the system is based on solving systems of Presburger formulae. Their later work considers properties about the values in containers [14], a language with references [15], space bounds for object-oriented languages [16], and applying similar techniques to assembly programs [17]. Vasconcelos has also studied these inference systems in order to

[1] http://homepages.inf.ed.ac.uk/bcampbe2/depth-analysis/

produce a sized types analysis and heap and stack space bounds for Hume [18]. One of the main advantages of a sized-types analysis over our approach is that the size information can be reused for other analyses.

The use of Presburger solvers in these systems raises concerns about efficiency, but it is unclear whether the generated constraints may include arbitrary formulae, or if they are limited to some easily solved subset. The reports cited above suggest that they are reasonable in practice. Similarly, certification using these systems may require the verifier to perform some constraint solving, whereas for our system they need only perform some simple arithmetic after reconstructing the typing using suitable hints.

A different approach which can yield non-linear bounds is to use recurrence solvers to deal with recursive functions, such as Debray and Lin's execution time analysis for logic programs [19], and Vasconcelos and Hammond's analysis [20]. The power of these analyses is dependent on the power of the recurrence solver used. Recent work by Albert et al. has tackled this by producing a specialised recurrence solver for dealing with cost equations [21].

9 Conclusions

We have presented a new stack space analysis similar in concept to the Hofmann-Jost heap space analysis, but with more structure to enable richer, more precise bounds.

Further work on the system could include enhancing the language (perhaps using the existing Hofmann-Jost extensions mentioned above), removing the need for users to provide the structure for type signatures and the containers extension outlined in Section 6. It would also be interesting to apply these techniques to heap space by devising a suitable replacement for the LET rule, especially as we may be able to regain the fine-grained form of plus-contraction when considering total space bounds.

References

1. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL '03: Proceedings of the 30th ACM Symposium on Principles of Programming Languages, New Orleans, ACM Press (2003)
2. O'Hearn, P.: On bunched typing. Journal of Functional Programming **13**(4) (2003) 747–796
3. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile resource guarantees for smart devices. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004. Volume 3362 of LNCS., Springer-Verlag (2005) 1–26
4. Jost, S., Loidl, H.W., Hammond, K.: Report on stack-space analysis (revised). Deliverable D05, The Embounded Project (IST-510255) (2007)
5. Campbell, B.: Prediction of linear memory usage for first-order functional programs. In: Trends in Functional Programming. Volume 9. (2008) To appear.

6. Tarjan, R.E.: Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods **6**(2) (1985) 306–318
7. Campbell, B.: Type-based amortized stack memory prediction. PhD thesis, University of Edinburgh (2008)
8. Jost, S., Loidl, H.W., Hammond, K.: Report on heap-space analysis. Deliverable D11, The Embounded Project (IST-510255) (2007)
9. Jost, S.: Amortised Analysis for Functional Programs. PhD thesis, Ludwig-Maximilians-University (2008) Forthcoming, provisional title.
10. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis (for an object-oriented language). In Sestoft, P., ed.: Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems. Volume 3924 of LNCS., Springer-Verlag (2006) 22–37
11. Reistad, B., Gifford, D.K.: Static dependent costs for estimating execution time. In: LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming, New York, NY, USA, ACM Press (1994) 65–78
12. Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space: towards embedded ML programming. In: ICFP '99: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming, ACM Press (1999) 70–81
13. Chin, W.N., Khoo, S.C.: Calculating sized types. Higher Order and Symbolic Computation **14**(2-3) (2001) 261–300
14. Chin, W.N., Khoo, S.C., Xu, D.N.: Extending sized type with collection analysis. In: PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial Evaluation and Semantics-based Program Manipulation, New York, NY, USA, ACM Press (2003) 75–84
15. Chin, W.N., Khoo, S.C., Qin, S., Popeea, C., Nguyen, H.H.: Verifying safety policies with size properties and alias controls. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, New York, NY, USA, ACM Press (2005) 186–195
16. Chin, W.N., Nguyen, H.H., Qin, S., Rinard, M.: Memory usage verification for OO programs. In: Static Analysis, 12th International Symposium (SAS 2005). Number 3672 in Lecture Notes in Computer Science, Springer-Verlag (2005)
17. Chin, W.N., Nguyen, H.H., Popeea, C., Qin, S.: Analysing memory resource bounds for low-level programs. In: ISMM '08: Proceedings of the 7th international symposium on Memory management, New York, NY, USA, ACM (2008) 151–160
18. Vasconcelos, P.: Space Cost Analysis Using Sized Types. PhD thesis, University of St Andrews (2008)
19. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. ACM Transactions on Programming Languages and Systems **15**(5) (1993) 826–875
20. Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In Trinder, P., Michaelson, G., Peña, R., eds.: Implementation of Functional Languages. Volume 3145 of Lecture Notes in Computer Science., Springer-Verlag (2004) 86–101
21. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In Alpuente, M., Vidal, G., eds.: Static Analysis: 15th International Symposium (SAS 2008). Volume 5079 of Lecture Notes in Computer Science., Springer-Verlag (2008) 221–237